Available online at www.sciencedirect.com

ScienceDirect

Computational Geometry 38 (2007) 4–15

Computational Geometry
Theory and Applications

www.elsevier.com/locate/comgeo

Almost-Delaunay simplices: Robust neighbor relations for imprecise 3D points using CGAL

Deepak Bandyopadhyay *, Jack Snoeyink [1]

Department of Computer Science, CB# 3175 Sitterson Hall, University of North Carolina,
Chapel Hill, NC 27599-3175, USA

Received 1 March 2006; received in revised form 2 November 2006; accepted 15 November 2006

Available online 16 February 2007

Communicated by B. Gärtner and R. Veltkamp

Abstract

This paper describes a new computational geometry technique, almost-Delaunay simplices, that was implemented for 3D points using CGAL. Almost-Delaunay simplices capture possible sets of Delaunay neighbors in the presence of a bounded perturbation, and give a framework for nearest neighbor analysis in imprecise point sets such as protein structures. The use of CGAL helps us tune our implementation so that it is reasonably fast and also performs robust computation for all inputs, and also lets us distribute our technique to potential users in a portable, reusable and extensible form. The implementation, available on http://www.cs.unc.edu/~debug/software is faster and more memory efficient than our prototype MATLAB implementation, and enables us to scale our neighbor analysis to large sets of protein structures, each with 100–3000 residues.

Keywords: Delaunay triangulation; Almost-Delaunay; Tessellation; Nearest neighbor; 3D; Imprecision; Robust; Protein structure

1. Motivation and definitions

The Delaunay tessellation (DT) is a geometric structure that defines a nearest neighbor relation on a set of points in space (known as *sites*), using an exact geometric criterion—the "empty sphere test" [1]. The DT has been used in the analysis of protein structure,[2] among other applications, for detecting pockets and cavities [3,4], scoring packing interactions and distinguishing native proteins from artificial decoy structures [5–7], and detecting patterns of local structure [8].

In applications that deal with protein data, point coordinates are known only imprecisely—factors such as experimental errors and protein flexibility introduce small changes in the point coordinates that can produce different sets of Delaunay simplices.[3] So, a natural question arises: whether analyses based on DT are stable and robust under changes

---

* Corresponding author. Present address: Johnson & Johnson Pharmaceutical R&D, 665 Stockton Drive, Exton, PA 19341.
E-mail addresses: debug@cs.unc.edu (D. Bandyopadhyay), snoeyink@cs.unc.edu (J. Snoeyink).

[1] Portions of this research were supported by NSF grant 0076984.

[2] For an introduction to protein structure see [2].

[3] In d dimensions, a k-simplex, $k \leqslant d$ is defined by $k+1$ affinely independent points; thus tetrahedra are 3-simplices.

0925-7721/$ – see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.comgeo.2006.11.003

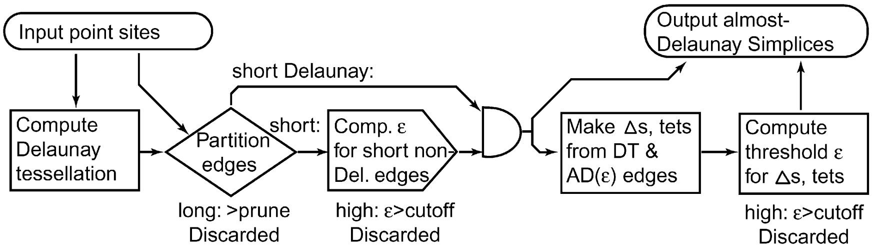
Input point sites
Compute Delaunay tessellation
Partition edges
short Delaunay:
short:
Comp. ε for short non-Del. edges
long: >prune Discarded
high: ε>cutoff Discarded
Make Δs, tets from DT & AD(ε) edges
Compute threshold ε for Δs, tets
high: ε>cutoff Discarded
Output almost-Delaunay Simplices

Fig. 1. Processing AD edges then AD simplices.

to the input coordinates. To answer this question, we defined the *almost Delaunay simplices* [9] that give possible neighbors under a bounded perturbation ϵ of the input.

Definition 1 *(Almost-Delaunay).* For a finite set of point sites P and $\epsilon \geqslant 0$, a *perturbation by* ϵ adds a vector $|v_i| \leqslant \epsilon$ to each $p_i \in P$. The set of *almost-Delaunay simplices* $AD(\epsilon)$ contains a k-tuple $S \subset P$ iff there exists a perturbation by ϵ producing $S_\epsilon \subset P_\epsilon$ such that S_ϵ has a circumscribing sphere containing no points of P_ϵ.

Each k-tuple, for $1 \leqslant k \leqslant d+1$, is an almost-Delaunay simplex for some minimum value of ϵ, denoted its *threshold*. The Delaunay simplices are those with threshold zero.

We related computation of the almost-Delaunay (AD) threshold to a variant of a minimum-width annulus problem, and proved properties of almost-Delaunay simplices [9] that we specialize here for tetrahedra in 3D:

(1) Local minima of the threshold correspond to annuli defined by 5 points in general position, with at least 2 on the inner sphere and 2 on the outer. From these 5 points we get 3 tetrahedra by considering all pairs from the sphere that has 3 points. A slab (annulus with infinite center and radii) is a special case defined by only 4 points.
(2) Points on the inner sphere form a Delaunay simplex.
(3) The center of the minimum-width annulus for a simplex S can come from the intersection of the Voronoi diagram of all points and the furthest-point Voronoi diagram of the points in S.

We then described an algorithm based on these properties. We identified two parameters arising from biological constraints that help us speed up our algorithm: the *edge length prune*, i.e. the maximum edge length between two neighboring sites, typically 10 Å; and the *threshold cutoff*, i.e. the maximum perturbation allowed, 0.5–2 Å depending on the type of perturbations of interest. Thus, we begin by computing the thresholds for almost-Delaunay edges, and from them, for triangles and tetrahedra as in Fig. 1.

In this paper, we describe our implementation of the AD threshold computation for edges, triangles and tetrahedra in 3D, which is written using CGAL. We describe the reasons for choosing CGAL, list the algorithm that is implemented, and give a minimal user manual for the resulting `AlmDel` software. We discuss some design decisions, unexpected issues that arose during development, our solutions, and the things we learned in the process. We report on verification and timing comparison with the previous implementation. Finally, we sum up the contributions of CGAL to our development efforts, and show that the use of CGAL helped achieve a fast and robust implementation of the almost-Delaunay simplices that greatly facilitated their application to protein structure analysis on a large scale.

2. CGAL implementation

We had built MATLAB prototypes of the almost-Delaunay simplex computation in 2D and later in 3D. The code for computing almost-Delaunay simplices from 3D points in MATLAB was very fast for small input sizes (100–250 points); it aggressively used MATLAB's vectorized operations instead of `for`-loops to compute on entire matrices at once, and thus trade time for space. MATLAB provides access to advanced geometric computations such as Delaunay triangulations and Voronoi diagrams through QHull [10], and provides many other functions that help accelerate the development of scientific and computational applications.

As we tried to compute AD simplices for proteins on a large scale, we found that the MATLAB implementation became unreasonably slow and ran out of memory. The implementation could not handle inputs that arose in practice, such as large protein chains represented by one point per residue (typically 1000 points) or medium-length chains with all atoms (typically 3000 points). To improve speed and memory utilization, we completely rewrote the code in C++ using CGAL.

2.1. Reasons for choosing CGAL

At the time we decided to implement our algorithm in C++, we had two clear choices of implementations to use:

(1) *CGAL* (Computational Geometry Algorithms Library, [11]), a C++ library containing high performance and robust implementations of many standard geometric data structures (e.g. Delaunay Triangulations, Convex Hulls, Polyhedral Surfaces, Arrangements), and the algorithms that operate on them (e.g. updation, insertion, deletion, point location).
(2) *QHull*, an implementation of the QuickHull algorithm [10] that computes convex hulls, Voronoi diagrams and Delaunay triangulations in all dimensions, and is fairly quick and robust in two and three dimensions.

Since the almost-Delaunay simplex algorithm usually operates on geometric primitives and does not need any computations more complex than convex hulls and Delaunay triangulations, QHull seemed a viable choice at first. Below, we describe some of the features of CGAL that prompted us to choose it for our implementation.

The CGAL library makes heavy use of the C++ Standard Template Library [12–14] and the Generic Programming paradigm [15–17]. The library is separated into three parts:

(1) The *kernel* [11,18] contains all the numerical and geometric types in two, three and general dimensions, and operations for creating them and testing them (constructions and predicates).
(2) The *basic library* packages computational geometry data structures and algorithms implemented using them.
(3) The *support library* contains extensions of STL developed for CGAL, and classes for visualization and input/output support.

CGAL prominently uses a technique from generic programming known as *traits classes*. All data structures and algorithms in the CGAL basic library are parameterized by template arguments, including arguments called *traits* that specify the interface between the data structure or algorithm and the data types or geometric primitives it operates on. Most default traits classes in CGAL are written in terms of the types and classes provided in the CGAL kernel; hence in most cases one plugs in the kernel as a traits class directly, and defines custom kernels parameterized using different number types, coordinate systems and other aspects of geometric behavior.

Traits classes allow the programmer to swap number types, data structures and algorithms in and out to achieve the optimum speed and robustness with minimal changes to the application code. We knew that achieving the correct tradeoff between robustness, accuracy and speed for our implementation was important, and required varying data types and even algorithms to find what worked best. The availability of traits classes convinced us to choose CGAL instead of QHull [10] for computing Delaunay triangulations and convex hulls.

Further, the CGAL code was modular and there were examples demonstrating the use of individual modules of the basic library. Data structures used to implement functionality (like the Triangulation Data Structure [19]) were well abstracted into an implementation layer. Data structures at all levels could be easily customized through traits classes, and were easy to navigate using a variety of access methods. The documentation on CGAL assured us that it had a tried and tested kernel, supplying efficient implementations of most of the operations that are possible on geometric objects such as points, lines and planes. By contrast, QHull's interface was less modular and harder to understand, and required the programmer to read and modify global variables; the authors recommended that users of QHull run the binary and communicate with it via input and output files rather than interface with its functions. Since QHull does not provide a complete geometric kernel, we would have had to re-implement more of the basic geometric operations, running the risk of incorrect implementation or mishandling of degenerate cases.

On the flip side, there was a steeper learning curve associated with learning CGAL, STL and generic programming techniques than there may have been with other libraries; this knowledge was needed to understand, tweak and debug CGAL implementations of geometric primitives, predicates and algorithms, and later to achieve robustness and speedups.

In Section 3 we discuss the lessons learned from the CGAL implementation, and in Section 4 we sum up the benefits of using CGAL and the reasons why its use was crucial to the success of our implementation.

2.2. *Algorithm implemented for 3D points*

The steps for the AD computation in 3D using CGAL are below:

- Generate list of short non-Delaunay edges, which are the potential AD edges, from the proximity graph (further described in Section 3).
- Consider any such edge, $\overline{pq}$.
  1. Compute the *candidate centers* of a minimum-width annulus, which are vertices of the intersection of the Voronoi diagram with the bisector plane of $\overline{pq}$. Infinite edges of the Voronoi diagram are candidate centers of slabs and are stored as directions. We use Brown's *lifting* technique [20], which involves computation of the lower convex hull in dual space.
  2. Evaluate the width of the annulus at each candidate center, which has p and q on the outer sphere and three points on the inner sphere (two for slabs).
  3. Find the minimum width over all candidate centers. Half of this width is the AD threshold for edge $\overline{pq}$.
  4. Retain the edge as *valid* if its threshold is less than or equal to the cutoff, otherwise remove it from the proximity graph.
  5. Retain candidate centers with half annulus width $\leqslant$ the cutoff (and candidate centers connected to them by Voronoi edges) for the next stage of computation.
- Output the list of valid AD edges and their thresholds.
- For each valid AD edge, $\overline{pq}$:
  - Generate possible AD triangles by querying the proximity graph for points r that are near both p and q.
  - For each point r:
    1. Select from $\overline{pq}$'s candidate centers, those whose annulus contains r. This is a linear half-plane constraint on the candidate centers, since all centers closer to r than to p or q would lie on the side facing away from p and q of the line equidistant to all three points.
    2. Generate new candidate centers at the intersection points of edges between candidate centers and the constraint, denoted *constraint cuts*. The existence of minimum-width annuli at constraint cuts is the reason we stored candidate centers with high threshold that were connected by edges to ones with half annulus width $\leqslant$ cutoff.
    3. Compute the annulus width at constraint cuts. p, q, r are on the outer sphere and the two points of the Voronoi edge are on the inner sphere.
    4. Find the minimum annulus for edge $\overline{pq}$ from among candidate centers satisfying the constraint generated by r, and constraint cuts of r.
    5. Record the threshold if it is less than cutoff.
  - The AD tetrahedron algorithm is similar. We generate the possible AD tetrahedra by querying the proximity graph for two points r and s that are near each other and also near both p and q. Then we generate two constraints (for r and s) and look for the minimum annulus among candidate centers that satisfy *both* constraints.
- The threshold for $\triangle pqr$ is the minimum of the threshold of $\overline{pq}$ constrained by r, $\overline{pr}$ constrained by q, and $\overline{qr}$ constrained by p. Similarly the threshold for a tetrahedron is the minimum constrained threshold over all the edges.
- Up to this point, the algorithm has produced all AD triangles and tetrahedra that have at least one non-Delaunay edge. In 3D there are AD triangles and tetrahedra with all edges Delaunay, which arise when three Delaunay tetrahedra share an edge whose vertices (say p and q) lie on opposite sides of the plane of the three other vertices (a, b, c); then $\triangle abc$ and tetrahedra $abcp$ and $abcq$ are almost-Delaunay. We enumerate these triangles and tetrahedra during a traversal of the Delaunay tessellation, and compute their thresholds separately for efficiency. For any such configuration of 5 points (a, b, c, p, q) the minimum width annulus for $\triangle abc$ and tetrahedra $abcp$ and $abcq$ is defined by only these 5 points, and has either (a, b),(b, c),(c, a) or (a, b, c) as furthest neighbors and the remaining points as closest neighbors.

2.3. *Brief user manual*

The almost-Delaunay software package (AlmDel) was released in June 2004, after about three months of development and five months of use and fixing bugs. In June 2005 it was described in a BioGeometry Newsletter article, and a permanent software download page was set up at http://biogeometry.cs.duke.edu/software/almdel. It contains about 4000 lines of C++ code written by the first author of this article. CGAL and Boost libraries needed for compilation are available at www.cgal.org. The source code has been released under a UNC Open Source Public License (similar to CGAL's QPL [21]), and two precompiled binaries are available for Windows and Linux. The ADCGAL binary computes almost-Delaunay edges, triangles and tetrahedra, while the ADedgeCGAL binary is specialized to compute almost-Delaunay edges only; this is the version that we use to build almost-Delaunay edge graphs [22]. Both programs take three parameters:

```
ADCGAL [<cutoff>] [<prune>] [<inputfile>]
ADedgeCGAL [<cutoff>] [<prune>] [<inputfile>]
```

Input Parameters: Parameter names above are enclosed within angle brackets and should be evaluated by substituting the parameter value. Explanation of the input parameters is as follows:

- `<cutoff>` is the threshold cutoff, i.e. the maximum perturbation allowed in the input data, in Å or the same units as the input coordinates. Typically it is between 0.1 and 2 Å for protein structure data. The default value of `<cutoff>` is 2.0 when it is not specified.
- `<prune>` is the edge length prune, i.e. the maximum length of any edge allowed in a simplex, also in Å or the input coordinate units. Typically it is chosen between 8 and 15 Å for protein structure data. The default value of `<prune>` is 10.0 when it is not specified.
- `<inputfile>` is the prefix of a file inputfile.out that contains the input point coordinates, one 3D point per row. By default, points are taken from standard input if this parameter is not specified; thus the `ADCGAL` program can read from an input pipe.

Output Format: The output of ADCGAL comprises indices of the edges, triangles and tetrahedra that are Delaunay, and those that are almost-Delaunay with positive thresholds up to the cutoff. The output is stored in text files named using the prefix `<inputfile>` as follows:

- `<inputfile>.del`, `<inputfile>.del3` and `<inputfile>.del4` store the indices of Delaunay edges, triangles and tetrahedra, one per row.
- `<inputfile>.AD`, `<inputfile>.AD3` and `<inputfile>.AD4` store the indices of almost-Delaunay edges, triangles and tetrahedra having positive threshold, along with their thresholds, one per row. Points are indexed in the C style, starting at 0 for the first point and going up to $n-1$ for the last (nth) point. The indices of points that comprise each Delaunay/AD edge, triangle and tetrahedron are sorted in increasing order, and the simplices are output in lexicographical order of their index vectors.

The `ADedgeCGAL` binary produces only files containing Delaunay and AD edges.

3. Lessons from CGAL implementation: Issues and design decisions

We now describe our CGAL experience, in particular the design decisions that were necessary to try to achieve speed and robust computation at the same time. Many of the issues and work-arounds described here occurred at the time we first implemented the software between October 2003 and May 2004, using CGAL version 3.0. CGAL at the time was still evolving from a library that worked its best when exact number types were used, to one that would work well in applications such as ours that need the speed of floating-point computation. As we predicted, work-arounds for many of the issues we faced were incorporated into the next major release of CGAL, version 3.1 in December 2004; thus, our experience, among others, has actually helped shape the growth of CGAL. We have indicated this in the following discussion.

Proximity data structures: We decided to use two different structures for proximity information, one to store the Delaunay tetrahedra with short edges, and another to store all short edges and non-Delaunay short edges. We considered the alternative of using algorithms available in CGAL for nearest neighbor searching, but since proximity is calculated only once, and we needed a way to remove pairs of points from the proximity relationship if their AD threshold is above the cutoff, we decided to store proximity in a graph structure. We used the Boost Graph Library (www.boost.org) to define a graph with nodes containing site indices and edges stored in an adjacency list representation. We encapsulated the graph in a class `ADPointNeighbors`, with methods to list all short edges and to find points or pairs of points near a short edge. *Update*: *Boost is now distributed with CGAL 3.1.*

The short Delaunay edges, triangles and tetrahedra were encapsulated into another class `MyDelaunay` that uses CGAL's `Delaunay_triangulation_3` for implementing 3D Delaunay tessellations. `ADPointNeighbors`, the proximity information class holds a reference to `MyDelaunay` and uses it to enumerate the Delaunay and non-Delaunay short edges. `MyDelaunay` uses a boolean state stored in each tetrahedron and each of its faces to provide traversals of only the short edges and the triangles and tetrahedra containing them. An earlier implementation that deleted cells (tetrahedra) containing long edges from the triangulation led to precondition violations and instabilities in the traversal, since the iterators for traversing a triangulation were not written to handle triangulations with missing cells.

Fast and robust computation of 3D convex hull using Delaunay insertion and static filters: We started out using the `Cartesian<double>` kernel for all computations since we were concerned about speed; exact computation was not a priority since the original MATLAB code also used floating-point computation, protein coordinates are imprecise, and they can be perturbed to remove degeneracies. We faced problems with the convex hull computation in CGAL using the `Convex_hull_3` class that implements the QuickHull algorithm [10]. We tried a sequence of kernel types and implementation tweaks that did not work, as below:

- `Convex_hull_3` using floating point arithmetic was faster than MATLAB's version of QHull, but crashed or went into an infinite loop for several common inputs.
- Computation using *exact arithmetic* was orders of magnitude slower than MATLAB.
- *Dynamic filters* implemented in the CGAL `Filtered_kernel` [23] provide exact evaluation of predicates using floating point number types and interval arithmetic; exact arithmetic evaluation is forced only if the interval of the result from floating point evaluation contains 0. The use of the `Filtered_kernel` made predicates robust, but the `Convex_hull_3` code still crashed since it depended on *exact construction* of a plane containing three points, on which an orientation test was failing.
- Implementation of a plane class that stores the three defining points and tests orientation against them using a determinant, instead of explicitly computing the plane equation. Now the code did not crash, but the use of `Filtered_kernel` left it still about 2–4 times slower than the MATLAB implementation.
- *Static filters* are a technique for exact evaluation of predicates nearly as fast as floating point evaluation, wherein the interval bounds to force exact evaluation of each predicate are computed at compile time or runtime using properties known about the input data, such as the range of input points or the number of bits needed to represent them. `Static_filters` is an undocumented CGAL kernel available in CGAL releases 3.0 and 3.1; it is planned to be merged into the `Filtered_kernel` class in later releases. Since `Convex_hull_3` uses a predicate (`has_on_positive_side_3()`) for which no static filtered version was available in CGAL 3.0, we could not speed up the convex hull computation any further while keeping it robust. We anticipate that this problem will be fixed in a future release of CGAL.

Thus, static filters offered the only hope for both fast and robust convex hull computation, but they could not yet be applied to 3D convex hull computation. To make use of static filters that have been written for the Delaunay tessellation [24], we computed the convex hull by *incremental Delaunay insertion*—running a DT point location step for each new point, and inserting only the points found to lie outside the convex hull of the current DT. In the end we report all faces on the convex hull of the DT (i.e. adjacent to an infinite face) as the convex hull. After this modification, the AD edge code using static filtered floating point computation was not much slower than the MATLAB code using floating point computation, as shown in Table 1.

Genericity through function objects: Function objects, also called *functors*, are generalizations of C function pointers in the Standard Template Library; they are objects that declare `operator()` and hence can be used like functions.

Table 1
Time taken by two steps of the AD computation for a few protein chains, on a P4 2.0 GHz with 512 MB of memory. Cutoff was 2.0 Å and edge length prune was 10 Å, except for the last chain which has coordinates for all atoms, closer together than representative points, where cutoff was 0.5 Å and prune was 6.0 Å

PDB ID	# pts	#nonD edges	AD edges (sec)		ADtri/tet (sec)	
			Matlab	CGAL	Matlab	CGAL
1ab8A	115	252	2.1	1.1	3.3	0.6
1jkeA	145	479	4.4	2.6	7.0	1.5
1bjfA	181	474	4.5	3.5	8.5	1.3
1jmkC	222	691	7.4	6.1	11.1	2.4
1g6aA	266	980	11.0	10.3	18.7	4.0
1c8bA	320	1024	11.9	13.6	20.8	4.5
1crzA	397	1557	20.0	25.1	41.3	8.4
1lj8A	481	1604	21.6	30.1	51.3	9.0
1m2oA	718	2491	40.5	72.1	136.4	19.1
1gaxA	862	2903	53.5	97.8	208.1	27.6
1iw7C	999	3390	108.0	140.1	307.8	34.3
1d2rA	2563	31537	nomem	3408	nomem	1193

Some advantages of function objects over function pointers are cleaner syntax of declaration and the ability to store state within a function. Function objects turn functions into first-class objects much like in functional programming languages, and enable their use in multiple instantiations, inheritance and polymorphism. They also facilitate the generic programming paradigm by enabling functions to operate on generic data and to themselves modify the behavior of templated classes, STL algorithms and other function objects to which they are template parameters. Function objects have many uses in CGAL, such as for implementing predicates, traversing data streams, and converting one type of data to another. The STL paradigm of using algorithms and function objects to do much of the monotonous work without the chance of error played a major role in making our code efficient and reusable, and was a major redesign from the MATLAB implementation. We made use of many of CGAL's function objects in our implementation, and also wrote function objects to encapsulate the following operations:

- the projection operation involved in lifted Voronoi and candidate center computation
- bounding box and centroid computations on an iterator range or a container of points
- mapping an iterator range of point objects to their vertex index numbers
- a constraint equation object to test ranges of candidate center points (including centers at infinity, stored as rays) against a half-plane constraint, reporting candidate centers that satisfy the constraint and edges between two centers that cross the constraint
- a sequence indexer to index a sequence container such as an array or a vector with multiple subscripts
- a sequence minimizer to find the minimum value in a container over a set of valid indices.

In our enthusiasm to make these function objects do everything, we gave many of them constructors that take an iterator range and call the STL algorithm `for_each` to apply the same function operation on the elements of that range (by passing it `*this`). However, we needed to pass the state reached during execution on the range back to the calling instance using a method call, since the function object used is copied by value and the state will otherwise be lost. In this example of code from a function object constructor, `out()` is a method that returns the value of state variable `result`. The first statement fails, while the second works.

```
// op() updates result in temporary copy... changes lost!
std::for_each(indices_list.begin(), indices_list.end(), *this);
// get updated result from copy using method
result=std::for_each(indices_list.begin(), indices_list.end(),
   *this).out();
```

Balancing library component use with custom development: We found that achieving a balance between using sophisticated existing packages and writing custom code helped speed our development process. For example, since our definition of minimum-width annulus does not require it to contain all points but only points of the simplex, we did not use or modify CGAL's `Minimum_annulus_d` package. Instead we implemented the lifting technique [20]. Available functions for projecting a 3D point onto a plane and lifting it to a paraboloid were unsuitable:

- `Plane_3.projection(Point_3)` did not transform coordinates to place the origin on the plane of projection and basis vectors perpendicular to it.
- `Plane_3.to_2d(Point_3)` was supposed to do what we wanted, but it treats plane coordinates with the world coordinate origin, rather than a point on the plane, as a reference point, and returns 2D coordinates that are scaled and do not preserve distance between the 3D points.
- `Construct_lifted_point_3` lifted to a plane rather than a paraboloid.

Hence, we hand-coded both the projection and lifting operations in a function object.

Also, we chose to calculate the fairly simple furthest point Voronoi regions (bisector plane between 2 points, and lines equidistant from 3 points), rather than use code for kth-order Voronoi diagrams such as Julia Flötotto's prototype listed on the CGAL site.

Degeneracies creep in: Though we had assumed that our point sets were degeneracy-free, we found that degenerate configurations for some predicates and constructions did arise. Some examples of degeneracies we had to deal with are as follows:

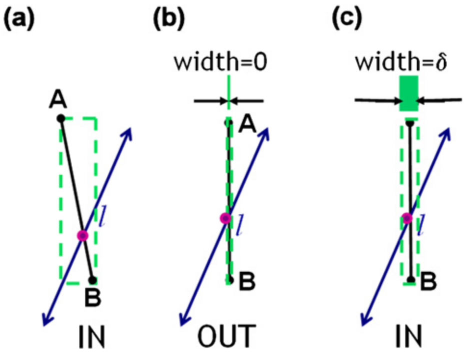

Fig. 2. (a) The 2D intersection computation between a line l and a segment $\overline{AB}$ is computed by intersecting l and the line through $\overline{AB}$ and checking if the intersection point lies inside the bounding box of $\overline{AB}$. (b) Intersection between a vertical segment and a line fails with floating point arithmetic since the intersection point of the two lines lies outside the bounding box of the segment that has zero width. (c) Increasing the width of the bounding box by an infinitesimal amount δ ensures that the intersection is detected.

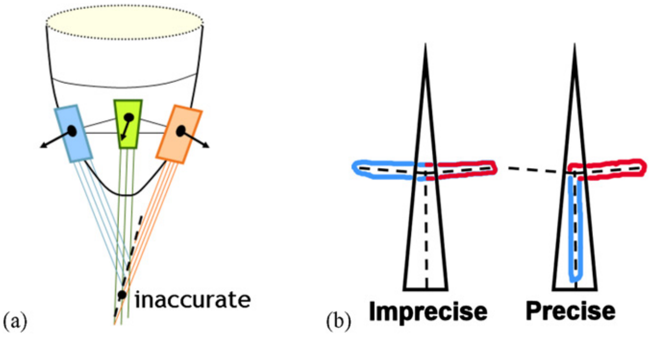

Fig. 3. (a) Intersection of three tangent planes computed by intersecting the leftmost plane and the rightmost plane to get a line, and then the line and the middle plane to get a point. This intersection point can be inaccurate if the first two planes are almost parallel, in the same way as in (b) the circumcenter of a very thin triangle can be computed with much more precision by intersecting the bisectors of a long side and the short side than by intersecting the bisectors of both long sides.

- The standard 2D Segment–Line intersection code in CGAL is implemented as a Line–Line intersection test, and reports the intersection point only if it lies within the rectangular bounding box defined by the segment. This fails for vertical or horizontal segments or those that are nearly vertical or horizontal, since the computed intersection point of the two lines lies outside the bounding box of the segment, as shown in Fig. 2. Similar problems affect the 2D Segment–Line, Segment–Ray, Ray–Line and Line–Line intersection tests that are used directly or indirectly by our code. Andreas Fabri on the CGAL development team helped us modify the code for these 2D intersection tests to widen the bounding box and thus avoid this problem. *Update*: *code for robust 2D intersections was added in CGAL 3.1.*
- We had two problems with robust 3D intersection of 3 planes. The first problem arises with three points that lie on a plane perpendicular to the bisector plane of two furthest neighbors, so that when projected on it they are collinear and when lifted their tangent planes do not intersect in a point. An imprecise intersection test on three such planes may find that an intersection point exists, but the resulting annulus with that candidate center has a negative width that is impossible and must be rejected.
- The second problem is with loss of precision in exact evaluation of the intersection point even when it exists. The CGAL operations and example code provided with CGAL version 3.0 for intersection point of three planes intersects two of the planes to find a line, and then the line and the third plane to find the intersection point, as below:

  ```
  CGAL::Object int_obj=CGAL::intersection(pl0,pl1);
  if (CGAL::assign(int_ln,int_obj))
     CGAL::Object int_obj2=CGAL::intersection(pl2,int_ln);
     if (CGAL::assign(int_pt, int_obj2))
        return int_pt;
     else throw "intersection point of 3 planes not a point";
  else throw "intersection point of 2 planes not a line";
  ```

  This method can fail to find the correct intersection point due to loss of precision in evaluating the first intersection line between planes that are nearly parallel. Such cases often arise while computing the intersections of three lifted tangent planes, when the first and second planes correspond to points that are nearly diametrically opposite with respect to the origin of the plane of projected points. This situation is depicted in Fig. 3(a), and is analogous to the 2D circumcenter computation of a thin triangle shown in Fig. 3(b) where intersecting the two nearly parallel bisectors gives a very imprecise answer while intersecting one of them with the bisector of the bottom edge gives a precise answer.

  The intersection of three planes can be computed robustly using floating-point arithmetic, if it exists, by solving the three plane equations using Cramer's rule:

  $$\begin{aligned}
  &a_x^1 \cdot x + a_y^1 \cdot y + a_z^1 \cdot z = c_1\\
  &a_x^2 \cdot x + a_y^2 \cdot y + a_z^2 \cdot z = c_2\\
  &a_x^3 \cdot x + a_y^3 \cdot y + c_z^3 \cdot z = c_3\\
  &\mathbf{A}\mathbf{x} = \mathbf{c} \Rightarrow \mathbf{x} = \mathbf{A}^{-1}\mathbf{c} = \big(\|\mathbf{A}_x\|/\|\mathbf{A}\|, \|\mathbf{A}_y\|/\|\mathbf{A}\|, \|\mathbf{A}_z\|/\|\mathbf{A}\|\big)
  \end{aligned}$$

  Above, $\mathbf{A}_x$, $\mathbf{A}_y$ and $\mathbf{A}_z$ are copies of matrix $\mathbf{A}$ with the first, second and third columns substituted by $\mathbf{c}$. All the above determinants should be evaluated as robustly as possible by subtracting one of the rows from the others. If the resulting determinant $\|\mathbf{A}\|$ is zero or nearly zero, the evaluation is retried after subtracting a different row. This is similar to the robust evaluation of predicates as proposed by Shewchuk [25]. *Update*: *a function for computing intersection of three planes was added in CGAL 3.1, but it has the same problems as before with floating point robustness, since it relies on a sequence of pairwise intersections.*
- Finally, the classical degeneracies do appear in some protein coordinates. Proteins represented by all atom coordinates contain coplanar atoms in the peptide bond and planar sidechains, and certain decoys built on a cubic lattice have cospherical atoms. In these cases we apply a random perturbation to the input of magnitude much smaller than any intended cutoff value, which changes the AD thresholds of simplices at most by a correspondingly small value.

As the CGAL project evolves, users find new applications and provide feedback, many of the tricks we used in our implementation will no longer be needed, and the functionality of many external libraries we used will be readily available in CGAL components.

3.1. Correctness, time complexity and comparison

We verified that our algorithm is implemented correctly by comparing the almost-Delaunay edge, triangle and tetrahedra thresholds output by it with the MATLAB implementation (which was itself tested against a brute-force implementation). The test set for the comparison comprised over 400 protein structures, run at different values of the cutoff and prune parameters. The results were found to match in every case.

We have shown earlier [9] that our algorithm takes $O(n^2 \log n)$ time for calculation of AD edges and $O(n^2)$ for calculation of AD tetrahedra in typical proteins, with some assumptions about the data being well packed that are valid for protein structures. The times taken by the implementation for 12 proteins are shown in Table 1. We observe that for computing edges, CGAL using filtered computation was not much slower than an optimized MATLAB implementation using floating point. For triangles and tetrahedra, CGAL was an order of magnitude faster, partly because the MATLAB implementation was complex and not fully optimized. Both stages of the MATLAB version ran out of memory for input data with 1000–3000 points, while CGAL did not.

4. Benefits of using CGAL

In this section, we briefly summarize our observations of how the choice to use CGAL benefited the development of our package. This discussion picks up from Section 2.1 where we mention why CGAL was initially chosen for this project.

- The flexibility of CGAL's kernel and traits classes and the variety of data structures, function objects and algorithms available allowed us to quickly experiment with a number of representations and methods to achieve fast and robust computation. For example, we used Delaunay insertion to compute convex hulls fast and robustly.
- The final performance of the static filtered, robust CGAL implementation was comparable with that of the floating-point MATLAB version for computing almost-Delaunay edges, while CGAL was an order of magnitude faster than MATLAB for computing almost-Delaunay triangles and tetrahedra.
- The reliability of the kernels and library routines allowed us to concentrate debugging efforts on logic errors in our own code, once the initial problem of achieving fast and robust floating-point computation with CGAL had been solved.
- The availability of comprehensive code examples and demos, and steady support from the CGAL team and the user support forum made the task of learning CGAL a lot easier once we had got past the initial stages.
- By using CGAL, the project took on some of the agility, versatility and high quality that is typical of CGAL code. CGAL code for tasks such as Delaunay triangulation and convex hulls is highly optimized and yet elegant. This makes the AlmDel package itself fairly high quality, and readily adaptable and extensible.
- CGAL is cross-platform, working on several platforms and compilers; this enables our code also to work on several platforms, making it accessible to a larger community of users. We have tested it on Windows (Visual C++. NET 2003, Intel Compiler, cygwin gcc), and on RedHat and other distributions of Linux (gcc-3.3.x).
- The ready availability of implementations of advanced computational geometry methods in CGAL will help in the future development of AlmDel and other related and derived packages that may need access to such methods.

The availability of CGAL was critical to the success of this project, since the application of almost-Delaunay simplices for analysis of protein structures had hit a bottleneck with the MATLAB implementation running out of memory on inputs with about 1000 points. The use of almost-Delaunay simplices in further analyses depended on the ability to compute them for proteins represented by 1000 or more points, and to process large families of $\sim$200 proteins or even the entire Protein Databank (PDB) of $\sim$10000 non-redundant proteins in a reasonable time. After the CGAL implementation was complete, we were able to develop these large-scale applications with great ease. Subsequently, we demonstrated the use of the almost-Delaunay simplices for many applications, such as:

- Scoring the packing of over 10000 models (with chains 50–500 residues long) submitted to the CASP5 protein structure prediction competition [26].
- Analyzing conformational change and identifying flexible residues in systems as complex as the Ca^{2+}-ATPase pump (994 residues) and the GroEL chaperonin (14 subunits with 518 residues each) [27].
- Building graph representations of protein structures using almost-Delaunay edges, for use in a subgraph mining algorithm to find packing patterns common to $\sim$200 protein families of 4–400 proteins each [28], and using these patterns to infer the function of new proteins from their structure [29].

5. Limitations and future work

While profiling the current implementation, we found that it wastes significant time in converting the points on the convex hull computed by Delaunay insertion, into a polyhedral surface, using the *Polyhedron_incremental_builder_3* class; on the average, 10–20% of the time taken to compute the convex hull. This step was necessary to interface with code that used CGAL functions to traverse a polyhedral surface returned by *Convex_hull_3*. Some of these functions, such as computing boundary half-edges that correspond to slabs, were difficult to write for a data structure not based on half-edges. Still, rewriting the code to traverse the surface of the DT would remove this overhead.

The implementation has migrated to a series of compilers and platforms as it got more complex through its development, and right now it uses CGAL-3.1 or higher, and has been compiled with Microsoft Visual C++ 7.1 and 8.0 (.NET 2003 and 2005) and the Intel compiler version 7.1 on Windows, and with `gcc` 3.3.x on cygwin and Redhat Linux 9.0.

While a dataset with 3000 points is large for a protein structure, almost-Delaunay simplices are not limited to analyzing protein structures. Among other areas, we foresee their use in computer graphics and terrain modeling, where researchers routinely study models with millions of points. Out-of-core techniques, data partitioning, hierarchical schemes, parallel processing and algorithmic improvements may help scale the implementation to larger numbers of points.

Though our framework allows computation of the almost-Delaunay simplices in dD, we have a complete implementation only in 3D since the driving problem of analyzing protein structure is in 3D. A 2D implementation is planned. We plan to make almost-Delaunay simplices available as a CGAL extension package, and pursue further algorithmic improvement (make it output sensitive, incremental and kinetic).

6. Conclusion

In this paper, we have shared the experience of implementing a complex computational geometry algorithm, almost-Delaunay simplices, in CGAL. CGAL helped us create a practical implementation of our algorithm in 3D that can process datasets with 100–5000 points in between a few seconds and an hour on a 2.0 GHz computer. CGAL was essential for this project to succeed, since it provided a stable and powerful set of tools for geometric representation and robust computation, which were highly parameterizable and could be used together in many different combinations to achieve the goal of a fast and robust implementation. The successful completion of this implementation helped us expand the scope and capabilities of our protein structure analysis studies.

Acknowledgements

We thank David O’Brien, resident STL and CGAL expert, and members of the CGAL developer community for help getting started and later for insightful suggestions and solutions. In particular we thank Andreas Fabri who pinpointed the convex hull robustness problem and wrote the fix for robust plane construction and code for Delaunay-insertion convex hull; Sylvain Pion for help with static filters and triangulations; and Radu Ursu for help with the Microsoft compiler.

References

[1] B. Delaunay, Sur la sphère vide. A la memoire de Georges Voronoi, Izv. Akad. Nauk SSSR, Otdelenie Matematicheskih i Estestvennyh Nauk 7 (1934) 793–800.

[2] C. Branden, J. Tooze, Introduction to Protein Structure, Garland Publishing Inc., New York, 1999.

[3] J. Liang, H. Edelsbrunner, P. Fu, P. Sudhakar, S. Subramaniam, Analytical shape computing of macromolecules II: Identification and computation of inaccessible cavities inside proteins, Proteins 33 (1998) 18–29.

[4] J. Liang, H. Edelsbrunner, C. Woodward, Anatomy of protein pockets and cavities: Measurement of binding site geometry and implications for ligand design, Protein Science 7 (1998) 1884–1897.

[5] C.W. Carter Jr., B.C. LeFebvre, S. Cammer, A. Tropsha, M.H. Edgell, Four-body potentials reveal protein-specific correlations to stability changes caused by hydrophobic core mutations, J. Molecular Biol. 311 (4) (2001) 625–638.

[6] R. Singh, A. Tropsha, I. Vaisman, Delaunay tessellation of proteins, J. Comput. Biol. 3 (1996) 213–222.

[7] B. Krishnamoorthy, A. Tropsha, Development of a four-body statistical pseudo-potential to discriminate native from non-native protein conformations, Bioinformatics 19 (2003) 1540–1548.

[8] H. Wako, T. Yamato, Novel method to detect a motif of local structures in different protein conformations, Protein Engineering 11 (1998) 981–990.

[9] D. Bandyopadhyay, J. Snoeyink, Almost-Delaunay simplices: Nearest neighbor relations for imprecise points, in: ACM-SIAM Symposium on Discrete Algorithms, 2004, pp. 403–412.

[10] C.B. Barber, D.P. Dobkin, H. Huhdanpaa, The QuickHull algorithm for convex hulls, ACM Trans. Math. Softw. 22 (4) (1996) 469–483.

[11] S. Hert, M. Hoffmann, L. Kettner, S. Pion, M. Seel, An adaptable and extensible geometry kernel, in: Proc. Workshop on Algorithm Engineering, in: Lecture Notes in Comput. Sci., vol. 2141, Springer-Verlag, 2001, pp. 79–90.

[12] N.M. Josuttis, The C++ Standard Library: A Tutorial and Reference, Addison Wesley, 1999.

[13] D. Vandevoorde, N.M. Josuttis, C++ Templates: The Complete Guide, Addison Wesley, 2002.

[14] S. Meyers, Effective STL: 50 Specific Ways to Improve the Use of the Standard Template Library, second ed., Professional Computing Series, Addison Wesley, 2001.

[15] M.H. Austen, Generic Programming and The STL: Using and Extending the C++ Standard Template Library, Professional Computing Series, Addison Wesley, 1999.

[16] A. Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied, AW C++ in Depth Series, Addison Wesley, 2001.

[17] H. Brönnimann, L. Kettner, S. Schirra, R. Veltkamp, Applications of the generic programming paradigm in the design of CGAL, in: M. Jazayeri, R. Loos, D. Musser (Eds.), Generic Programming—Proceedings of a Dagstuhl Seminar, in: Lecture Notes in Comput. Sci., vol. 1766, Springer, 2000, pp. 206–217.

[18] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, S. Schönherr, The CGAL kernel: A basis for geometric computation, in: M.C. Lin, D. Manocha (Eds.), Proc. 1st ACM Workshop on Appl. Comput. Geom., in: Lecture Notes in Comput. Sci., vol. 1148, Springer-Verlag, 1996, pp. 191–202.

[19] J.-D. Boissonnat, O. Devillers, S. Pion, M. Teillaud, M. Yvinec, Triangulations in CGAL, Comput. Geom. Theory Appl. 22 (2002) 5–19.

[20] K.Q. Brown, Geometric transforms for fast geometric algorithms, PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1980.

[21] Trolltech AS, Q public license v1.0, 1999.

[22] J. Huan, W. Wang, D. Bandyopadhyay, J. Snoeyink, J. Prins, A. Tropsha, Mining protein family specific residue packing patterns from protein structure graphs, in: Eighth Annual International Conference on Research in Computational Molecular Biology (RECOMB), 2004, pp. 308–315.

[23] H. Brönnimann, C. Burnikel, S. Pion, Interval arithmetic yields efficient dynamic filters for computational geometry, in: SCG ’98: Proceedings of the Fourteenth Annual Symposium on Computational Geometry, ACM Press, New York, 1998, pp. 165–174.

[24] O. Devillers, S. Pion, Efficient exact geometric predicates for Delaunay triangulations, in: 5th Workshop on Algorithm Engineering and Experiments (ALENEX 03), Baltimore, MD, 2003.

[25] J.R. Shewchuk, Robust adaptive floating-point geometric predicates, in: SCG ’96: Proceedings of the Twelfth Annual Symposium on Computational Geometry, ACM Press, New York, 1996, pp. 141–150.

[26] J. Moult, K. Fidelis, A. Zemla, T. Hubbard, Critical assessment of methods of protein structure prediction (CASP)-round V, Proteins 53 (6) (2003) 334–339, Suppl.

[27] D. Bandyopadhyay, A geometric framework for robust nearest neighbor analysis of protein structure and function, PhD thesis, University of North Carolina, Chapel Hill, NC, 2005.

[28] J. Huan, D. Bandyopadhyay, W. Wang, J. Snoeyink, J. Prins, A. Tropsha, Comparing graph representations of protein structure for mining family-specific residue-based packing motifs, J. Comput. Biol. 12 (6) (2005) 657–671.

[29] D. Bandyopadhyay, J. Huan, J. Liu, J. Prins, J. Snoeyink, W. Wang, A. Tropsha, Structure-based function inference using protein family-specific fingerprints, Protein Science 15 (6) (2006) 1537–1543.